A Service Dependency Modeling Framework for Policy-based Response Enforcement

Nizar Kheir[1,2], Hervé Debar[1], Frédéric Cuppens[2], Nora Cuppens-Boulahia[2] and Jouni Viinikka[1]

[1] France Télécom R&D Caen, 42 Rue des Coutures BP 6243, 14066 CAEN, France
{nizar.kheir, herve.debar, jouni.viinikka}@orange-ftgroup.com
[2] Télécom Bretagne, 2 rue de la Chataigneraie, 35512 Cesson Sévigné Cedex, France
{frederic.cuppens, nora.cuppens}@telecom-bretagne.eu

Abstract. The use of dynamic access control policies for threat response adapts local response decisions to high level system constraints. However, security policies are often carefully tightened during system design-time, and the large number of service dependencies in a system architecture makes their dynamic adaptation difficult. The enforcement of a single response rule requires performing multiple configuration changes on multiple services. This paper formally describes a Service Dependency Framework (SDF) in order to assist the response process in selecting the policy enforcement points (PEPs) capable of applying a dynamic response rule. It automatically derives elementary access rules from the generic access control, either allowed or denied by the dynamic response policy, so they can be locally managed by local PEPs. SDF introduces a *requires/provides* model of service dependencies. It models the service architecture in a modular way, and thus provides both extensibility and reusability of model components. SDF is defined using the Architecture Analysis and Design Language, which provides formal concepts for modeling system architectures. This paper presents a systematic treatment of the dependency model which aims to apply policy rules while minimizing configuration changes and reducing resource consumption.

1 Introduction

Intrusion Detection Systems (IDSes) have been recently superseded by Intrusion Prevention Systems (IPS), which add the capability to passivate the threat in addition to detecting and reporting. IPSes are widely used as local control points which take only limited actions (e.g. closing a connection, killing a process, etc.). The major weakness of those IPSes is their static behavior, which relies on predefined mappings between intrusive behaviors and suitable response actions. The taxonomy in [1] thus confirms the need for more complex and dynamic response mechanisms. Cuppens et al. propose in [2] a reaction workflow which links the local response decisions to the higher level of security policy. They state that local response decisions should be assisted by global decisions managed at the policy level. In [3], Debar et al. provide a comprehensive approach for managing intrusion response at the policy level using contextual security policies.

Intrusion response is specified using contextual access control rules which are triggered when their associated threat contexts are activated. The policy-based response architecture in [3] separates the response instantiation process which triggers response rules from the response decision process. While the instantiation process is sufficiently detailed in [3], contributions for the decision process remain sparse. This paper completes the architecture in [3] by defining a service modeling framework which enables the decision process to automatically select local enforcement points able to apply a dynamic response rule.

The decision process maps policy instances into concrete actions applicable on local Policy Enforcement Points (PEPs). [4] proposes a derivation process which translates high level policies into local firewall actions, but it only manages network policies. The use of application services provides more granularity for the specification and application of response policies. A service may be configured with accurate access control rules to the data it manages. It thus enables accurate response applications not always possible at the network layer. Moreover, service dependencies may provide several alternatives for the application of response policies. The access to a dependent service may be modified through the reconfiguration of the access to its antecedent service. Unfortunately, the use of service dependencies for automated response is still, at the best of our knowledge, restrained to static mappings. The lack of a formal representation of those services and their dependencies is a major reason. This paper combines policy-based response with topological information about services and their dependencies. Services endowed with access control capabilities (ACLs, Application Firewalls, configuration files, etc.) are considered as PEPs. The Service Dependency Framework (SDF), a formal framework for modeling services and their dependencies, is defined in this paper. It assists the decision process in deriving local accesses to antecedent services from the generic access to the dependent service which is either allowed or denied by the response policy. The decision process analyzes those accesses with respect to PEPs capabilities. It selects the optimal set of PEPs capable of applying the security rule.

The paper is structured as follows. Section 2 summarizes the state of the art, including the presentation of the policy-based response process, the need for a SDF and description of related work. Section 3 presents the service dependency model. Section 4 defines the framework for building the dependency model. Section 5 provides a systematic treatment of the SDF by the decision process. Section 6 concretely implements the SDF on a mail delivery testbed.

2 State Of the Art

2.1 Policy-Based Intrusion Response

Access control policies include permission and/or prohibition rules which apply to subjects when they intend to perform actions on objects. Some rules specify requirements which apply during normal operation, they form the *operational* policy. Others apply in case of security threats, they form the *threat* policy. The switching between operational and threat policies is driven by contextual

constraints. We specify contextual policies using the Organization Based Access Control (OrBAC) Model [5]. This paragraph recalls the OrBAC concepts we need in this paper (see [6] for details).

The OrBAC Model uses the abstract triplet *(Role, Activity, View)* instead of the concrete triplet *(subject, action, object)* when defining access control policies. The concept of *Role* was first introduced by the RBAC model [7]. A *Role* is a set of subjects which have the same permissions. OrBAC adds *Activities* and *Views* as abstractions of *actions* and *objects* respectively. An Activity (e.g. access data) is an operation implemented by some actions (e.g. *get* and *retr* commands for http and pop3 protocols respectively). These can be grouped within the same activity for which we may define a single rule. A *View* is a set of objects that possess the same security-related properties so they can be accessed in the same way. Abstracting *objects* into *Views* avoids the need for writing one rule for each of them. OrBAC introduces contexts [5] which add conditions under which a certain rule can be applied. OrBAC uses four predicates:

- *empower (subj, Role)*: subject *subj* is empowered in the role *Role*.
- *consider (act, Activity)*: action *act* is implemented in the activity *Activity*.
- *Use (obj, View)*: object *obj* is used in the view *View*.
- *hold (ctxt, subj, act, obj)*: context *ctxt* is active for the triplet $(subj, act, obj)$.

A security rule is expressed as *Sr (Decision, R, A, V, Context)*[3]. When *context* is active, R's request to perform the activity A on the View V is submitted to the decision *Decision*. An example of an OrBAC security rule is: *Sr (Prohibition, User, login, internal_Host, not_working_Hours)*. *User* is a role for any system user; *login* is the activity of connecting to a host; *internal_Host* is any host connected to the internal network and the context *not_working_Hours* is true outside working hours. *Sr* states that such an operation is prohibited outside working hours.

Reaction Policies in the OrBAC model are associated with threat contexts. A threat context is assigned to an intrusion class. It is activated when the associated intrusion is detected (e.g. DoS, Buffer overflow). As in [3], threat contexts are only activated for the concrete triplets described in alerts. For instance, a brute force attack from a certain address *addr* against an account *Acc* using the *login* service activates the *Brute_Force* context as follows:

Hold (addr, login, Acc, Brute_Force) ← *alert (Source, Target, Classification), Classification (Brute_Force), service (Target, login), Account (Target, Acc)*

The context activation may trigger policy rules associated with this context. These rules specify new security requirements appropriate for countering the detected threat. Threat contexts activation uses mappings from IDMEF alert attributes [8] onto concrete policy components. While the mappings are deliberately simplified in our example, they may introduce different granularities in order to consider different attack classes (e.g. a DDoS attack is managed differently than a targeted buffer overflow attack)[3].

[3] OrBAC associates organizations with security rules. To simplify, these are not made visible in this paper since we consider only a single organization

2.2 Policy Decision Models

The decision process contains two steps [4]. The first is architecture-dependent. It segments a response into elementary actions. The second step is component-dependent. It translates elementary actions into concrete configurations. [9] proposes a formal approach for the specification and deployment of network security policies. Unfortunately, it does not consider the overlying service architecture. The modeling of services and their dependencies provides means for a fine grained response application, which it is not always applicable on the network layer. Let's check for instance the following concrete response policy:
Prohibition (IP, HTTP/Get, retail_Appli), Permission (IP, HTTP/Get, mail_Appli)
where both applications are hosted on the same server. Applying this policy at the network layer is tedious since firewalls are less likely to have visibility over application data. While application layer firewalls are more appropriate, they are not always available on the server. A model-based analysis of service architectures may provide more suited alternatives for the application of such response policies. For instance, when the web server is accessible through traffic redirection from a remote proxy, a SDF links between the web service and the proxy service so that the decision process automatically selects the proxy for applying this policy.

A formal dependency framework which establishes the link between the access to an antecedent service and the access to its dependent service does not seem to exist. This paper provides a *requires/provides* model framework for service dependencies. This framework assists the Policy Decision Process (PDP) in tracing all the elementary accesses in order to access a certain data. As such, and when this access is prohibited, the PDP alters some elementary accesses in order to deny the prohibited access. Moreover, when the access is allowed, the PDP satisfies at least one single access path to the data. The decision process in this paper is different from the approach described in [10] in that it aims at finding the best suitable set of PEPs capable of applying a response, after and only after this response is selected. We first briefly describe existing dependency models and their usages before presenting our dependency model and its use.

2.3 Service Dependency Models and Applications

Existing dependency models An XML based dependency model is presented in [11]. This model provides a backend for building a dependency database, without providing a formal specification of service dependencies. [12] defines a dependency algebra for modeling dependency strengths. It separates the *Dependency* relation from the *Use* relation. It states that critical components should only use and not depend on non-critical components. In [13], a UML-based dependency model describes service dependencies in ad hoc systems. It focuses on the dependencies relevant to ad hoc collaborative environments. Moreover, a service dependency classification for system management analysis is provided in [14]. It separates between functional (implementation-independent) and structural dependencies (implementation-dependent).

Service Dependency Usages A cost-sensitive approach for balancing between intrusion and response costs is provided in [15]. A system map holding dependency information is used as a basis for deciding on response strategy. [16] proposes a function which evaluates intrusion response impacts using dependency trees. It allows a cost-sensitive selection of intrusion responses. Another cost-sensitive analysis of intrusion responses is presented in [17]. It uses dependency graphs instead of dependency trees. Service dependencies are also used for fault analysis [18], dependability analysis [19] and many other applications.

The existing dependency models such as graph [15,17,16] or class-based [13] models classify service dependencies using static attributes. These are often informally defined, and adapted to only specific system implementations. The adoption of those models still confronted to their expressiveness and the dependency characteristics they deal with. The decision process needs more than to know about the existence of a certain dependency and its strength. In order to derive elementary accesses to antecedent services, and to do it automatically, the decision process must be able to discern the access to the antecedent service through the access to its dependent services. In other terms, it must be aware of what data is required from the antecedent service, how, when and why is it accessed.

On the other hand, the SDF must enable the regrouping of elementary services into dependency blocks with well-defined interfaces. Those blocks can be implemented in other dependency blocks, and thus providing reusability of the dependency model. The SDF must also allow the abstraction of certain dependencies, and thus representing only the dependencies relevant for the application purposes. We have choosen to define the SDF using the Architecture Analysis and Design Language (AADL)[20]. AADL fulfills those requirements through the modeling of service architectures. In the following section, we summarize the main AADL concepts we use in this paper and present our SDF.

3 The Service Dependency Model

3.1 Using AADL to model the SDF

AADL has been released and standardized by the Society of Automotive Engineers. AADL provides formal modeling concepts for the description and analysis of application system architectures in terms of distinct components and their interactions. We privileged AADL over common modeling languages like UML because AADL provides more powerful features for modeling system runtime behaviors. AADL provides standardized textual and graphical notations for modeling systems and their functional interfaces. It has been designed to be extensible so that analyses that the core language does not support can be supplied. The extensibility in AADL is provided through the *Annex* extension construct.

SDF models user runtime behaviors when accessing the data provided by dependent services. It contrasts with most functional dependency models since it focuses on the data flows associated with the access to a dependent service rather than modeling its functional dependencies. This is a main concept in our

approach since policy-driven responses require PEPs to deny some of these data flows. We thus model services as abstractions, and these are decoupled from the concrete components which realize them. Our decision can be best motivated by the fact that concrete components only introduce functional dependencies which are not relevant in our approach. For instance, a web service is defined through its dependencies, independently whether it is implemented by apache2 server or windows web server. We use for this purpose AADL system abstractions (see section 3.2). AADL models dependencies using inter-component connections. AADL connnections reproduce the service topology. They allow modeling multiple service paths through the use of multiple connection paths to the same data. We also use AADL operational modes in order to represent the dependency sequencing during the workflow of the dependent service. We use the *AADL Error Model Annex* [21] which has also been standardized to add features for modeling the system behavior in the presence of faults. We use faults as model constructs in order to represent the behavior of a dependent service when it can not access to the antecedent service due to a response application. In the remaining of this section, we describe the main elements of our AADL dependency model.

3.2 Service and service dependency definition

We define a service as the implementation of an interface which *provides* data access to its users (e.g. Web service, IP service). A service often *requires* access to subsidiary data during its normal behavior. It is thus identified through the specification of its required and provided data accesses. We model an elementary service in AADL as a black box with specific *requires/provides* interfaces. Each interface enables a specific data access, either required or provided by the service (see Figure 1). We may add constraints between data required and provided by a service (e.g. the required account is the owner of the provided data). These are expressed as predicates assigned, when necessary, to the corresponding interfaces.

```
1   -- Implementation of elementary service --
2   system Service_Name
3    features
4     RF1: requires data access data_Set_r1;
5     ...
6     RFn: requires data access data_Set_rn;
7     PF1: provides data access data_Set_p1;
8     ...
9     PFm: provides data access data_Set_pm;
10  end Service_Name;
```

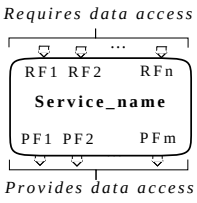


Fig. 1. Elementary Service definition

Service A depends on service B when A requires data access which is provided by B. A is the *dependent* service, and B is the *antecedent* service. The failure of B, due to an attack or a response, prevents it from providing the data required

by A. The proper behavior of A is thus conditioned by the proper behavior of B. Required data accesses enable dependency compliance check: A may never depend on a B if the data access provided by B is not required by A. However, a required data access does not necessarily imply the need for a dependency, because this access can be managed by the service itself. For instance, a mail delivery service requires access to user accounts. These can be managed locally by the service (passwords file), or remotely accessed through a directory service. Only the latter case implies a dependency for the directory service.

We model the dependency of service A to service B by connecting the *provides* interface of B to its complementary *requires* interface of A. The AADL model checks the compliance of this dependency by verifying that the access required by A corresponds to the access provided by B (see Figure 2).

```
1  system implementation Dependency_Model.A
2   subcomponents
3    A: system dependent;
4    B: system antecedent;
5   connections
6    const_AB: data access B.PF1 -> A.RF1;
7  end Dependency_Model.A;
```

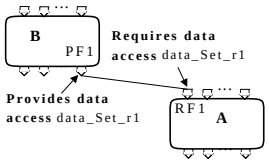


Fig. 2. Explicit Service Dependency Representation

3.3 Service Dependency Specification

The SDF specifies dependencies by modeling: the data exchanged in each dependency, the paths followed by these data, the sequencing of dependencies during the operation of the dependent service and the impact due to the unfulfillment of each dependency. We thus define the following dependency characteristics.

- *Dependency type* defines the path of the network flow, and describes the data assets exchanged between the dependent and the antecedent service.

- *Dependency mode* makes precise the occurrence of a dependency within the lifecycle and workflow of the dependent service.

- *Dependency Impact* evaluates the influence of the insatisfaction or degradation of the relation between antecedent and dependent services.

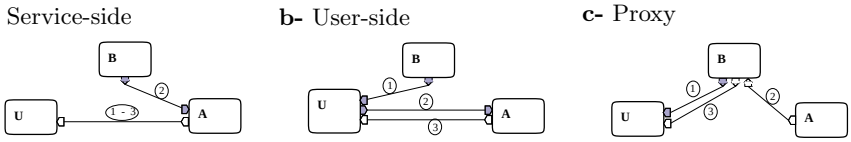
While these characteristics may be completed at a later time, we believe that they are the most relevant for our purpose of using the dependency model for assisting the decision process as described in section 2. In the remainder of this section, we discuss each attribute, and we show how it is modeled in AADL.

Service Dependency Types describe elementary paths followed by the data provided by the antecedent service. They only describe access paths for the direct dependencies of a service. Complete data paths, due to indirect dependencies (dependencies of the direct antecedents of a service), are automatically inferred from elementary access paths for each service as explained later in section 4.

A dependency type may be either *service-side*, *user-side* or *proxy* dependency.
- *Service-side dependency:* the dependent service initiates the interaction with the antecedent service. The user connects to the dependent service as if no dependency exists (see Figure 3-a).

- *User-side dependency:* the user obtain credentials from the antecedent service and present them to the dependent service. The connection is transparent for the dependent service (see Figure 3-b).

- *Proxy dependency:* the access path to the dependent service is intercepted by the antecedent service. No access path explicitly exists between the dependent service and its user during the dependency (see Figure 3-c).



White interfaces represent the data flow provided by the dependent service for its users.
Gray interfaces represent data flow provided by the antecedent service.
A is the dependent service, B is the antecedent service, and U is the user of the dependent service.

Fig. 3. Service Dependency Types

Service Dependency Modes describe the sequencing of dependencies within the lifecycle and workflow of the dependent service. We use AADL operational *modes* for modeling dependency sequencing. AADL *modes* are constructs which represent operational states of a component. Each mode illustrates an operational phase for the dependent service which is characterized by the need for a certain dependency. As such, the dependent service does not notice the failure and/or inaccessibility of the antecedent service unless the former reaches an operational mode where it requires the access to the data provided by the antecedent service. The transition into a dependency mode means that the dependent service has reached an operational phase where it requires access to the data provided by the antecedent service. The transition out of this mode means that the dependency is no longer required.

A service has four operational modes. These modes describe the lifecycle of this service. Every dependency mode exists necessarily in at least one of these operational modes. We shall first describe service lifecycle in AADL, and later we describe dependency sequencing during this lifecycle.

Service lifecycle holds four operational modes: Start, Idle, Request and Stop modes (see the associated AADL model in Figure 4). They are defined as follows:

- *start Mode* characterizes the launching period of a service. The process realizing the service is loading configurations and assets. The transition out of

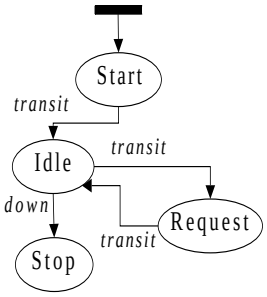
this mode occurs when the process is ready to receive user requests. Dependencies in start mode are one-time dependencies only required during service start-up.

- *Idle Mode* characterizes the period during which a service is waiting for incoming user requests. The transition out of this mode is initiated by a user request, or by a decision to stop the service. The dependencies in this phase are mainly functional dependencies not relevant for the purpose of this paper, but which can be further investigated as for impact evaluations (see section 7).

- *Request Mode* starts when the service receives a user request. It characterizes the in-line dependencies required in order to process this request. The transition from this mode occurs after the user connection is closed.

- *Stop mode* All the actions a service may take before stopping are considered as part of the stop mode.

The sojourn time in each operational mode varies according to service configurations. Transitions between operational modes may also vary for certain services. For instance, a service may start on a per-request basis. It therefore directly switches to the stop mode at the end of the request mode.




```
1  system implementation Dependent.instance
2   subcomponents
3     CStart: system op_State in modes (Start);
4     CIdle: system op_State in modes (Idle);
5     CRequest: system op_State in modes (Request);
6     CStop: system op_State in modes (Stop);
7   modes
8     Start: initial mode;
9     Idle: mode; Request: mode; Stop: mode;
10    Start -[CStart.transit]-> Idle;
11    Idle -[CIdle.transit]-> Request;
12    Request -[CRequest.transit]-> Idle;
13    Idle -[CIdle.down]-> Stop;
14 end Dependent.instance;
```

Fig. 4. Dependent Service Modes

Dependency sequencing Dependencies in each operational mode are invoked in a certain sequence related to the service behavior. These are defined as AADL operational sub-modes assigned to the components of each operational mode (lines 2-6 in Figure 4). We thus state dependencies within the lifecycle of the dependent service, and we determine the dependency sequencing within the same lifecycle phase. We obtain a Dependency Finite State Machine (DFSM) with sub-states. Dependencies appear in three possible sequences described as follows.

- *Stateless sequencing*: the satisfaction of the parent dependency is an obligation prior to the access to the child dependency. However, the former does not need to remain satisfied once the latter is accessed (Figure 5-a).

- *Statefull sequencing*: the parent dependency must remain satisfied as long as the child dependency is not satisfied yet (Figure 5-b).

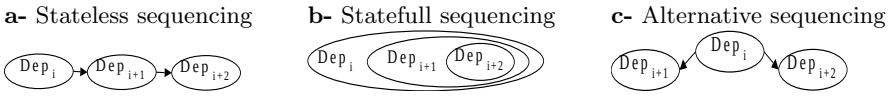


Fig. 5. Service Dependency Sequencing

- *Alternative sequencing*: characterizes redundant dependencies. The transition from the parent dependency leads to one child dependency (Figure 5-c).

Stateless and statefull sequencings express conjuctive dependencies. Alternative sequencing expresses disjunctive dependencies where only one alternative dependency is required. Each dependency mode is associated with a specific *require* interface (see Figure 1) which is connected to a specific antecedent service.

Service Dependency Impacts express the consequence of any degradation of the antecedent service, which alters the access to data required by the dependent service. The failure of a dependency alters the transitions between operational modes. This alteration is motivated by the fact that the failure of a dependency denies reaching its subsequent dependencies in case of no alternative dependency.

Dependency failure does not only alter the normal transition out of the failed dependency. It may also restrain the service to switch to another operational mode. For instance, a web server may switch to unsecure connections when the SSL service does not respond. We use the AADL error model annex to represent the impact of a dependency failure. Each service is attributed at least two AADL error states, which are normal and failure states. The impact of a dependency is expressed by constraining the transition out of a dependency to occur depending on the error state of the antecedent service. This is done by defining *Guard_Transition* properties which use error propagations. Error propagations are AADL constructs which notify the component at the remote end of a connection about the error state of the other component. We use *Error_Free* and *Failed* propagations which notify respectively an error free and a failed dependency states. Each dependency state may dispose of two transitions. The first is the normal transition, constrained by the satisfaction of the dependency. The second transition is optional. It is constrained by the unsatisfaction of the dependency.

The following example of a Mail Delivery Service (`MDS`) illustrates these specifications. `MDS` authenticates its users using `LDAP` service. Authenticated users are granted access to their remote mailboxes mounted using the `NFS` service. The accounts of connected users are locked in order to prevent simultaneous sessions. `MDS` unlocks an `LDAP` account after its corresponding user closes his/her opened session. The normal behavior of `MDS` is modeled in lines 1-6 of Figure 6.

The impact of the second `LDAP` dependency is stated as follows. Firstly, authenticated users cannot disconnect if the `MDS` cannot access to the `LDAP` service. The *Guard_Transition* in lines 11-12 states that the transition to the Idle phase (line 6) only occurs if the dependency is in the *Error_Free* state. Secondly, authenticated users remain blocked in the `NFS` dependency state as long as the second `LDAP` dependency is not restored (lines 7-10 of Figure 6).

```
modes
 LDAP1: initial mode;
 NFS: mode; LDAP2: mode; Idle: mode;
 T1: LDAP1 -[C1.transit]->NFS;
 T2: NFS -[C2.transit]-> LDAP2;
 T3: LDAP2 -[C3.transit]-> Idle;
 T4: LDAP2 -[C3.Failure_transit]-> NFS;
annex Error_Model {**
  Guard_Transition =>
  (RAccount[Failed]) applies to T4;
 Guard_Transition =>
  (RAccount[Error_Free]) applies to T3;
**};
```

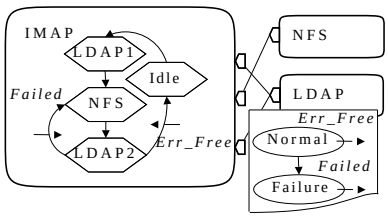


Fig. 6. Service dependency Impact

4 Dependency Model Framework

Section 3 has defined the service dependency characteristics managed using our approach. This section describes the steps for building a dependency model using our framework summarized in Figure 7. We use the Open Source AADL Tool Environment (`OSATE`)[4] which is a set of `Eclipse` plug-ins. `OSATE` maintains AADL models as XML-based files, which allows the reusability of the model.

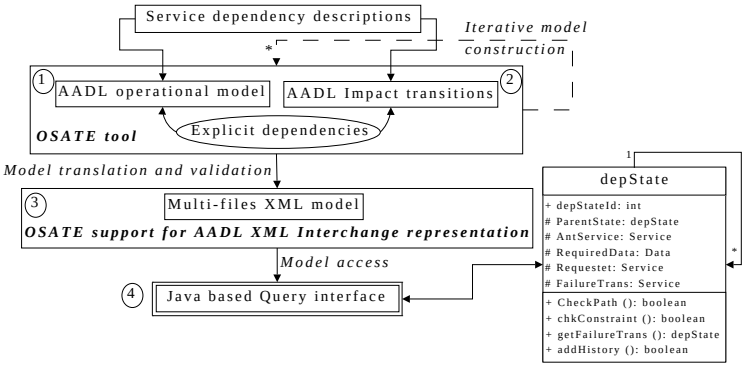


Fig. 7. Dependency Model Framework

The modeling framework is split into four steps. The user is intended to do the first two steps. The last two steps are automatically generated.

Step 1 consists of modeling the explicit dependencies of a service. Each service has a dedicated dependency model defined in an AADL package. Only explicit dependencies are represented. Antecedent services are considered as independent services, and therefore indirect dependencies are not represented.

[4] `http://la.sei.cmu.edu/aadlinfosite/OpenSourceAADLToolEnvironment.html`

Step 2 consists of modeling the dependency impacts. Failure impacts are specified as in section 3.3. Only the impacts of explicit dependencies are modeled. Indirect dependency impacts are infered from those of explicit dependencies.

The iteration over the first two steps consists of replacing antecedent services by the implementation of their composite dependency models. Antecedent services, previously used as abstract independent components, are replaced by instantiations of their dependency packages (see the case study for examples).

In **Step 3**, `OSATE` translates the AADL model into a multi-file XML model. Each package (i.e. elementary dependency model) is saved as an XML file expressed using the AADL XML Interchange format. This step is preceded by an automated model validation. `OSATE` checks the connections between model components. It flags inappropriate dependencies where a dependent service is made dependent of an antecedent service which does not provide its required data.

Step 4 is the implementation of a query interface which manages the access to the dependency model. This interface is queried for the dependencies of a specific service. We use the Java-based Document Object Model to explore the AADL/XML model. The query interface builds a Dependency Finite State Machine (DFSM) with substates in order to represent service dependencies.

The DFSM schema is illustrated in Figure 7. It summarizes all the dependency characteristics modeled in the first two steps. The attributes of a dependency state are (1) the antecedent service, (2) the required data (section 3.2), (3) the requester (dependency type), (4) the dependency impact, (5) the parent dependency and (6) the next dependency (dependency modes). Cyclic dependencies are discarded, and thus a dependency state cannot be a parent for another dependency state which points to the same service.

5 Service Dependencies: Application to Security

5.1 Using Services as Policy Enforcement Points

Deriving Concrete DFSM from Abstract DFSM Policy-based responses are expressed as (s, a, o) triplets. The SDF is queried for the DFSM of the service which implements the action a. It thus provides a DFSM which holds abstract components while the PDP receives concrete rules (see section 2.1). We thus need to derive a concrete DFSM using the abstract DFSM provided by the query interface. We associate abstract services with the concrete components which realize them using the predicate $realize(component, Service)$. It states that the service $Service$ is realized by the component $component$. The derivation process replaces the abstract service with its concrete implementation. It also derives concrete data instances from abstract data interfaces through the application of their associated constraints to the object o in the security rule (see section 3.2). A service may be realized by several component instances. The single abstract dependency is thus instantiated into several disjunctive concrete dependencies.

The derivation process follows the dependency sequencing in the abstract DFSM. It substitutes abstract components with concrete implementations. The

```
input  : Sr(s, a, o), DFSM
output: DFSM
curState = DFSM.start;
repeat
    if curState == DFSM.start then
        Only in the first iteration
        curState.Requester = Sr.s;
        curState.AntService = Sr.a;
        curState.RequiredData = Sr.o;
    else
        if curState.Requester == User then curState.Requester = Sr.s;
        else curState.Requester = subject.realize(subject, curState.Requester);
        curState.RequiredData = curState.RequiredData.chkConstraint(Sr.o);
        auxSr.s = curState.Requester; auxSr.a = curState.AntService;
        aurSr.o = curState.RequiredData;
        MakeTransClosure(auxSr, curState.getChilds());
        getChilds() Returns the sub state machine for the current state
    curState = curState.getNext();
until curState = DFSM.end ;
```

Algorithm 1: Transitive Closure

initial concrete components are provided by the concrete response rule. Subsequent concrete accesses are derived from the abstract DFSM and the concrete response rule. We use for this purpose the `MakeTransClosure` function of algorithm 1. It iteratively substitutes the abstract DFSM with a concrete DFSM using the concrete security rule delivered by the policy instantiation process.

Modeling Policy Enforcement Points The derivation of concrete elementary accesses is followed by a decision process. It aims to reconfigure elementary accesses so that the initial response access rule could be applied. In case of permission, the decision process satisfies at least a minimal set of dependencies. In case of a prohibition, it checks that no dependency path enables the prohibited data access. Access permissions are modified through the reconfiguration of PEPs which are modules associated with services. We therefore consider each service as a PEP having limited access control capabilities. This capability, when it exists, is limited to a specific class of subjects. It thus restrains the PEP capability to apply elementary access rules. For instance, firewall visibility is limited to network level information, it is not able to monitor user-level credentials.

A PEP is able to apply a security rule when (1) the subject in this rule belongs to the capability set of the PEP, (2) the service pointed by the action is managed by the PEP and (3) the object is a data provided by the service (this constraint is satisfied by the derivation process of algorithm 1)). The capability of a PEP depends on its concrete implementation (see examples in the case study). It is defined as a constraint which must be satisfied by the subject in the security rule. Services which do not have access control capabilities are assigned null capability sets. The PDP may select a certain PEP if the subject within the elementary concrete rule derived for this PEP belongs to its capability class. The PDP selects the optimal response set according to two criteria.

- A prohibition is applied the closer possible to the start state of the DFSM, in order to reduce resource consumption. This is motivated by the fact that

when the access is denied at the beginning of the DFSM, subsequent dependency accesses are denied, which contributes in reducing resource consumption.

- The PDP minimizes the configuration changes required for the application of a security rule by minimizing the services which need to be reconfigured.

Section 5.2 describes how we fulfill those requirements using our approach.

5.2 Selecting Policy Enforcement Points

S is the set of services obtained from the AADL model. We model the DFSM for the service s_{Dep} as $DFSM_{s_{Dep}} = \{S_a, T_a\}$ where $s_i \in S_a \subset S$ is an antecedent for s_{Dep} and $a_{ij} \in T_a \subset S \times S$ is a transition. A path p_{ij} is a sequence of adjacent transitions which lead from the dependency state s_i to the dependency state s_j. If this path does not exist then $p_{ij} = \phi$. For an input security rule, the PDP crosses $DFSM_{s_{Dep}}$. It searches the minimal set of dependencies which applies the security rule and reduces superfluous resource transactions. Algorithm 2 illustrates the behavior of the PDP. In case of a permission, the PDP searches for the dependency path which requires the least modifications (i.e. reconfigurations) in order to allow the access. The selected path is liberated in order to apply the input permission. In case of a prohibition, the PDP denies all dependency paths. When altering a dependency state, the PDP switches to the failure transition of this state and checks that it does not belong to a permissible path.

```
input  : Sr(Type, s, a, o)
output: List < s_i, Sr_i > Resp with s_i ∈ S
FSM_a = makeTransClosure(getDFSM(a), Sr);
dStart = FSM_a.start; dEnd = FSM_a.end;
if Type = Prohibition then
    foreach p_ij in FSM_a with (i=dStart) & (j=dEnd) do
        if chkRespHistory(p_ij) (returns False if the path has beed already intercepted)
        then
            curState = dStart;
            repeat
                curState = curState.getNext(p_ij); returns the next state on the path p_ij
                if chkCapability(curState) then
                    Resp.add(curState.AntService, curState.Sr);
                    curState.addHistory(curState.Sr); add Sr to the resp. history
                    auxPath = FSM_a.getPath(curState.getFailureTrans(), dEnd);
                    if (auxPath ≠ φ)∧(curState.getFailureTrans().parent ≠ Idle) then
                    p_ij ← auxPath;
            until curState = dEnd ;
else
    In case of permission, the PDP allows the path requiring minimum modifications
    minPath = null; minLength = Infinity;
    foreach p_ij in FSM_a with (i=dStart) & (j=dEnd) do
        curLength = 0;
        repeat
            curState = curState.getNext(p_ij);
            if !chkRespHistory(curState) then curLength ++;
        until curState = dEnd ;
        if curLength < minLength then  {minLength = curLength; minPath = p_ij;}
    allow(minPath); Liberates the path in parameter
```

Algorithm 2: Evaluation of the resulting impact transfer matrices

6 Case Study: E-mail Service

6.1 Testbed description

This section implements our dependency model for the example of an email service. The email testbed manages mailboxes using the `NFS` service. Local mail access is granted by both `IMAP` and `POP` services. Remote mail access is granted by a webmail service. The webmail application connects directly to the `POP` server, and indirectly to the `IMAP` server through an `IMAP` proxy which caches `IMAP` connections. Users are authenticated using the `LDAP` service.

The available PEPs are `ModSecurity`[5] which monitors the access to the webmail application, the super daemon `XInetd` which monitors access to the `IMAP` Proxy. The `LDAP` server monitors the access to user accounts and the `NFS` service monitors the access to the shared files using the `/etc/exports` file. The visibility of `XInetd` and `NFS` is limited to internal `IP` addresses. `ModSecurity` only manages external `IP` addresses. Finally `LDAP` manages its internal accounts.

6.2 Description of the Testbed AADL Model

Figure 8 illustrates the graphical AADL reprentation of the testbed dependency model. The main parts of the AADL textual representation are described in appendix A. We interpret in this paragraph the AADL code in appendix A.

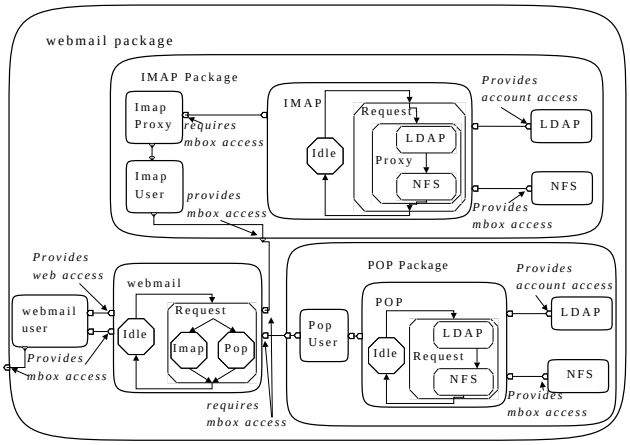


Fig. 8. TestBed AADL model

The `serviceDB` package (lines 1-13) contains the modeled services. `POP` service requires access to user accounts (line 4). It provides access to mailboxes (line

[5] `http://www.modsecurity.org/`

6) which are remotely accessed by the POP service (line 5). The webmail service is granted by a webmail application which must be accessible for webmail users (line 10). The webmail service recuperates mailboxes (line 9) and provides them to its users (line 11). The POP package (lines 14-35) provides mailbox access (lines 15-17). The LDAP and NFS (line 20-21) services are extracted from the serviceDB package. LDAP and NFS dependencies are service-side dependencies; they are both connected to the POP service (lines 23-24). They are in the request mode (lines 27-28) since they are accessed by the POP service while processing user requests. LDAP dependency is stateless because the access to user accounts is not required after authentication (line 29). Its failure alters the transition to the NFS dependency (lines 29&32). The failure of the NFS dependency initiates a transition to the Idle mode (lines 30&33). The modeling of POP and IMAP dependencies (The IMAP package is omitted for space limitations) gives two packages which provide mailbox access. We use these packages in order to model the webmail service (lines 40-41). The latter is granted by a webmail application. We model the access for webmail users to the webmail application through the connection in line 46. The webmail application provides access to mailboxes recuperated from the mail delivery services (lines 44-45). Lines 53-56 model the multiple access paths to the mail boxes using both POP and IMAP services. The access to the web service must be maintained as long as the connection to the mail delivery services is still required. The web dependency is thus a statefull dependency. POP and IMAP dependencies are modeled as substates of the web dependency (line 49).

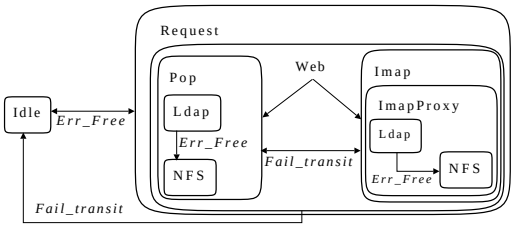


Fig. 9. Webmail Dependency FSM

The query interface generates a webmail DFSM (see Figure 9) which summarizes all dependency states and transitions (both normal and failure transitions).

6.3 The Use of Service Dependencies for Response Application

We demonstrate in this section the use of the SDF. We adopt the same mappings provided in [3], and we show that for the same abstract rule, the selected PEPs vary according to the mapping outcome. We prefer to use the simple response policy shown in listing 1.1 in order to show the use of our dependency model. This response policy requires that the attacker must be forbidden from accessing to the threatened data through the victim service.

Listing 1.1. Testbed Response Policy

```
1 -- The abstract response rule --
2 Sr (prohibition, att_Source, victim_Serv, target_Data, attack_Threat)
3 -- The Or-Bac Hold fact which transforms alerts into contexts --
4 Hold (Subject, Action, Object, Th_Context) :-
5   alert (Source, Target, description) &
6   map_Subject (Source, Subject) &
7   map_Action (Target.Service, Action) &
8   map_Object (Target, Object) &
9   map_Context (description, Th_Context).
```

The mapping functions in listing 1.1 are XSLTs which extract data from IDMEF alerts [8]. We implemented a prototype for algorithm 2. We simulated several attack instances and we observed the subsequent behavior of the PDP. In the remaining, we give four attack examples and the associated responses fired by the Policy Instantiation Engine and managed by the PDP using our prototype. Figure 10 summarizes the alerts received, the PDP behavior and the selected PEPs. It also shows the configurations automatically generated for each selected PEP according to its appropriate elementary access rule derived by the PDP. The attributes in italic are simple mappings from the associated access rules. These are generated by component-specific agents interfacing with each PEP.

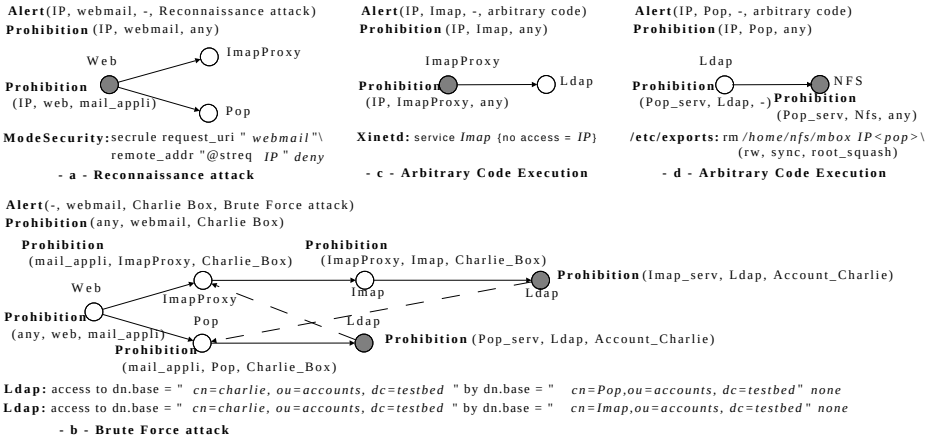


Fig. 10. Attack and Response Samples managed by the Testbed

Reconnaissance attack is generated by an external user who tries to find valid user IDs. The attacker does not have a valid account. The alert source thus lacks information about a known user. The response module is alerted about an IP address performing a reconnaissance attack against the webmail application. As in Figure 10-a, the PDP selects the first dependency state since the source specified in the elementary rule belongs to the PEP capability set.

Brute Force attack is account centric. The attacker has already acquired a valid account ID. He now tries to break the associated password. The alert notifies a brute force attack from a spoofed address against Charlie's mailbox. The

dependency selected in the former example can not be used since no IP address is selected. The PDP chooses to deny the access for both `POP` and `IMAP` servers to Charlie's account. The dashed arrows (Figure 10-b) are failure transitions followed by the PDP after it has altered their source dependency nodes.

Arbitrary Code Execution allows an intruder to execute arbitrary code on the target machine on the behalf of the exploited service. The threatened services are `IMAP` and `POP` respectively. The alerts respectively notify an `IMAP` and a `POP` threat. The selected DFSMs are those of `POP` and `IMAP` services. In case of `IMAP` service (Figure 10-c), the first dependency is selected since the source IP address belongs to the capability set of `Xinetd`. The `LDAP` dependency for the `POP` service can not be used since no `LDAP` account was instantiated by the transitive closure (Figure 10-d). The `NFS` service is found to be able to apply its elementary access control rule. It consists of unmounting mailboxes in the `/etc/exports` file. It is true that the decision process did not provide a solution which protects the `POP` server. However, a close look to the PEPs capabilities shows that such a solution at least protects the mailbox alteration following a successful attack.

7 Discussion and Conclusion

In this paper, we have presented a modeling framework for the services and their dependencies. The novelty of this framework resides in its ability to formally define dependency attributes, rather than assigning static dependency parameters as in most of the existing class-based models. The formal definition of dependency parameters provides a strong platform for the use of those dependencies for security management. This paper demonstrates that service dependencies can be used for more than only *a-posteriori* evaluation of intrusion response impacts, after these have been selected (although being an important challenge for the security research community). It describes an *a-priori* use of service dependencies, notably for the selection of suitable means to apply an intrusion response, if any. The efficiency of a response application is measured through its ability to satisfy the security requirements while pushing the response closer to the attacker and minimizing the configuration changes.

Limitations of this work include the separated treatment of responses and dependencies search. Firstly, the separated treatment of each response will be extended in order to consider the overall response policy. The optimal application of each response apart does not necessarily provide an optimal application of the response policy, as certain rules may overlap. However, since new response rules may be generated continuously, other problems must be considered such as the stability of the system. Secondly, the upward search for dependencies can be extended with a downward search (i.e. searching for dependents of a service) of dependencies in order to evaluate the impact of selected responses. Future work will focus on adding a third criterion for the selection of a candidate response, being its impact on other services. This will be seen as collateral damages since an antecedent service may have several dependent services other than the service explicitly designated in the intrusion response.

References

1. Stakhanova, N., Basu, S., Wong, J.: A taxonomy of intrusion response systems. Int. Journal of Information and Computer Security **1** (2007)
2. Cuppens, F., Cuppens-Boulahia, N., Bouzida, Y., Kanoun, W., Croissant, A.: Expression and deployment of reaction policies. SITIS Wkshp. Web-Based Information Technologies & Distributed Systems (2008)
3. Debar, H., Thomas, Y., Cuppens, F., Cuppens-Boulahia, N.: Enabling automated threat response through the use of a dynamic security policy. Journal in Computer Virology **3** (2007)
4. Cuppens, F., Cuppens-Boulahia, N., Sans, T., Miège, A.: A formal approach to specify and deploy a network security policy. Wkshp. Formal Aspects in Security and Trust (2004)
5. Cuppens, F., Miège, A.: Modeling contexts in the or-bac model. In: Proc. Int. Annual Computer Sercurity Application Conf. (2003)
6. Kalam, A.A.E., Baida, R.E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C., Trouessin, G.: Organization based access control. In: IEEE Int. Wkshp. Policies for Distributed Systems and Networks. (2003)
7. Sandhu, R.S., Coynek, E.J., Feinsteink, H.L., Youmank, C.E.: Role-based access control models. In: IEEE Computer. Volume 28. (1996) 38–47
8. Debar, H., Curry, D., Feinstein, B.: The intrusion detection message exchange format. Internet Draft, RFC 4765 (March 2007)
9. Preda, S., Cuppens-Boulahia, N., Cuppens, F., G.Alfaro, J., Toutain, L.: Reliable process for security policy deployment. Int. conf. Security and Cryptography (2007)
10. Papadaki, M., Furnell, S.M.: Informing the decision process in an automated intrusion response system. Information Security Technical Report **10** (2005)
11. Ensel, C., Keller, A.: Managing application service dependencies with xml and the resource description framework. In: Proc. Int. IEEE Symp. Integrated Management. (2001) 661–674
12. Ding, H., Sha, L.: Dependency algebra: A tool for designing robust real-time systems. In: Proc. IEEE Int. Real-Time Systems Symp. (2005)
13. Randic, M., Blaskovic, B., Knezevic, P.: Modeling service dependencies in ad hoc collaborative systems. In: Proc. Int. conf. EUROCON. (2005)
14. Keller, A., Kar, G.: Dynamic dependencies in application service management. In: Proc. Int. conf. Parallel and Distributed Processing Techniques and Applications. (2000)
15. Balepin, I., Maltsev, S., Rowe, J., Levitt, K.: Using specification-based intrusion detection for automated response. In: Proc. Int. Symp. Recent Advances in Intrusion Detection. (2003)
16. Toth, T., Kruegel, C.: Evaluation the impact of automated intrusion response mechanisms. In: Proc. Int. Annual Computer Sercurity Application Conf. (2002)
17. Jahnke, M., Thul, C., Martini, P.: Graph based metrics for intrusion response measures in computer networks. In: IEEE Conf. Local Computer Networks. (2007)
18. Gruschke, B.: Integrated event management: Event correlation using dependency graphs. In: Proc. Int. Wkshp. Distributed Systems. (1999)
19. Rugina, A.E., Kanoun, K., Kaâniche, M.: A system dependability modeling framework using aadl and gspns. Architecting Dependable Systems, LNCS 4615 (2007)
20. SAE-AS5506: Sae architecture analysis and design language. Int. Society of Automotive Engineers (November 2004)
21. SAE-AS5506/1: Sae architecture analysis and design language, error model annex. Int. Society of Automotive Engineers (June 2006)

8 Appendix A

This section summarizes the AADL textual representation for the email testbed.

```
package serviceDB -- Service database --
  public -- Only two sample services are presented --
    system POP  -- Implementation of the Pop service --
      features mb_Owner: requires data access dataDB::Account;
               R_mb: requires data access dataDB::mBox;
               P_mb: provides data access dataDB::mBox;
    end POP;
    system WEBMAIL -- Implementation of the webmail service --
      features R_mb: requires data access dataDB::mBox;
               R_api: requires data access dataDB::mailAPI;
               P_mb: provides data access dataDB::mBox;
    end WEBMAIL;
end serviceDB;
package Pop -- The implementation of the Pop dependency model --
  public System POP
          features P_mb: provides data access dataDB::mbox;
        end POP;
  private system implementation POP.instance
            subcomponents PopUser: system user;
                          Ldap: system serviceDB::Ldap;
                          NFS: system serviceDB::NFS;
                          Pop: system dependent.instance;
            connections data access Ldap.P_a-> Pop.mb_Owner;
                        data access NFS.P_mb-> Pop.R_mb;
                        data access Pop.P_mb-> PopUser.R_mb;
          end POP.instance;
          system implementation op_State.Request
            modes LDAP: initial mode; NFS: mode; Idle: mode;
                  T1: LDAP-[C1.transit]-> NFS;
                  T2: NFS-[C2.Fail_transit]-> Idle;
            annex Error_Model {**
              Guard_Transition => (mb_Owner[Error_Free]) applies to T1;
              Guard_Transition => (R_mb[Failed]) applies to T2; **};
        end op_State.Request;
end Pop;
package webmail
  public -- same as the Pop public interface --
  private system implementation webmail.instance
            subcomponents webmailUser: system user;
                          Imap: system Imap::IMAP;
                          Pop: system Pop::POP;
                          web: system serviceDB::Web;
                          webmail: system dependent.instance;
            connections data access Imap.P_mb -> webmail.R_mb1;
                        data access Pop.P_mb -> webmail.R_mb2;
                        data access web.P_api -> webmailUser.R_api;
                        data access webmail.P_mb -> webmailUser.R_mb1;
    end webmail.instance;
    system implementation op_State.web
    subcomponents C1: system op_State in modes (Idle);
                  C2: system op_State in modes (Pop);
                  C3: system op_State in modes (Imap);
    modes Idle: initial mode; Pop: mode; Imap: mode;
          T1:Idle-[C1.transit]->Pop;T2:Idle-[C1.transit]->Imap;
          T3:Pop-[C2.Fail_transit]->Imap;T4:Imap-[C3.Fail_transit]->Pop;
          T5:Pop-[C2.transit]->Idle;T6:Imap-[C3.transit]->Idle;
    annex Error_Model {**
          Guard_Transition => (R_mb2[Failed]) applies to T3;
          Guard_Transition => (R_mb1[Failed]) applies to T4;
          Guard_Transition => (R_mb2[Error_Free]) applies to T5;
          Guard_Transition => (R_mb1[Error_Free]) applies to T6; **};
    end op_State.web;
end webmail;
```